22028

RESEAU CYCLADES

SCH 517

September 1973

INWG Note # 50

NIC # 22028

NETWORK PROTOCOLS

Louis POUZIN

Institut de Recherche d'Informatique et d'Automatique (IRIA)

Rocquencourt, France

SUMMARY

Various data transfer capabilities are required for user traffic. The set of rules governing exchanges is called a protocol, which is a variety of language. Protocols use communications tools on which assumptions must be made. In particular they are not error free.

Protocol implementations are distributed machines whose components must be synchronized. Error detection and back-tracking are necessary functions, and require internal communication for control information.

At the lowest level one needs a basic message transfer capability insuring reliable communications over an unsafe medium. This can be achieved using named messages and acknowledgements, with a few other limited assumptions. Varieties of communication procedures are actually based on this simple scheme. Options arise in managing the message name space. A window technique reducesoverhead while allowing non sequential transfers.

User protocols implement various tools tailored to common requirements. There are discussions about the proper localization of functions in the system hierarchy. Server protocols are typical cases of distributed machines requiring a specialized user language, and an internal communication machinery. The Arpanet file transfer protocol is discussed.

Host protocols are a mixture of functions which can be put to work independently. They should be stripped down so as to provide just a basic communications layer allowing arbitrary extensions. Other approaches are possible.

This paper has been presented at the NATO International Advanced Study Institute, University of Sussex, Brighton, 9-15 September 1973.

Received at NIC 22-FEB-74

I. USER REQUIREMENTS

Programming a computer has a variety of acceptations. E.g. :

. Write a program in assembly language to develop a tape handler
. Use an access method to update an index-sequential file
. Write a COBOL program
. Sort a file using a library processor
. Submit a job from a remote card reader
. Login and type commands to a time-sharing system, then work with a conversational text-editor.

All these activities require using some kind of language, typically several, depending on which component or which level of system is to be dealt with. Each language is supposedly well defined, and supported by a gamut of tools : manuals, compilers, interpreters, debugging aids. In many cases components can be implemented in one language, handled with another, and used as a building block (macro, sub-routine, processor) in the context of yet another language. All that makes up the traditional resource environment of a computer system.

Computer networks have brought about an additional set of user needs, owing to the geographical dispersion and the heterogeneity of their resources. Furthermore, the practical implementation of appropriate user tools has to cope with specific problems resulting from present communications technologies. Investigations pursued so far generally reveal the following main problem areas :

a) Physical I-O ports between a computer (host) and a communications network must be limited to a small number (1 to 3). A multiplexing machinery is needed to funnel all host-to-host messages through a few I-O lines.

b) In order to exchange messages, host processes need some network-wide name space for addressing each other.

c) The communications network imposes a maximum message length. Longer items must be fragmented and reassembled.

d) Some messages may not reach their destination. Hence error control.

e) Unchecked message traffic may flood the receiving host. Hence flow control.

f) I-O streams should extend over the network. Hence the need for virtual links between processes.

g) File transfer between different hosts.

h) Job submission to a distant host.

j) Communications between various servers and terminals.

Working Size Type Area : 33 1/2 x 52 Picas (To be shot at 90%)
Final Size Type Area : 38 x 57 Picas
Final Trim Size : 6 1/2 x 9 inches (165 x 229 mm)

k) Data conversion and reformatting.

l) Distributed data base management.

m) User identification, accounting.

n) Information centers, and how to get help.

... etc...

There is no doubt that users would find overly cumbersome and inefficient to use for similar functions as many different languages as hosts. This is yet quite awkward to have a Babel on a single computer. Consequently network standards must be defined for the more popular functions, so that they use the same language on every host.

On a single computer instructions of a program are fed into a processor. In a network several processors may be involved simultaneously on different hosts. Conceptually they interact with one another as if in direct contact. However, some mechanisms are required to bridge the physical distance. Therefore additional network standards must be defined to reckon with inter-processor exchanges. They have been called "protocols", but they are also languages as we shall see.

In Arpanet protocols tend to be limited to the definition of an inter-processor language for a particular network usage. In Cyclades protocols tend to include also the definition of a user language network-wide.

II. COMMUNICATIONS PRINCIPLES

System structures :

When analyzing the functions of system components, it is customary to refer to predefined underlying mechanisms called the "lower level", with which one assumes a set of well defined relations. Similarly, each component is supposed to implement a set of functions to be invoked by a "higher level" in a well defined manner. This layered approach is now classical and needs no further comment in this context.

Confusion may arise when "high" and "low" are taken from a different realm, such as history, efficiency, sophistication, priority. E.g. the SITA High Level Network is actually a lower level structure to a message carrier service. Particularly, in dealing with message transmission, one frequently introduces another (outer) level of envelope, which only reflects another (inner)

level of system. These obvious remarks intend to dispel any persistent misunderstanding which sometimes degenerates into a religion war.

In purely hierarchical systems, any level sees only its next lower neighbor as being a simplified model of whatever happens to be built underneath, (Fig. 1). A frequent terminology refers to the higher level as the user, and the lower levels as the system, while the set of rules governing the mutual relationship is an interface. Communications between both levels is often straightforward, when they share a common store, and this function is not even identified as a relevant subject.

It may happen that "the system" is so constructed that it cannot be reached by "the user" without resorting to non-trivial mechanisms. Communications become a problem per se, and results in a more intricate structure. This is the case when physical components are geographically separate, but it also appears in multiprocessor computers (1). A communication level slips in between "user" and "system", (Fig. 2). Is it higher or lower than what ?

As emphasized previously, high and low levels are just conventional visions of structural relationship. They do not have to carry any connotation of physical or time dependencies. There cannot be any formal proof leading to a proper layered structure. In a subjective sense, a good one should simplify and help to understand, while conserving flexibility, autonomy, and efficiency in implementation. It comes as no surprise that interpretations vary widely about the fulfillment of such objectives. As for beauty or virtue it relates to personal background.

Having to deal with a communications level may be a technical necessity, but that should not blur the fact that the fundamental objective is to interface two levels, as simply as if there were no communications problem at all. In other words, our simple 2-level structure should not be defaced by an accidental component, though its presence must be recognized. This concern results in a 3-component structure, where "user" and "system" maintain their relationship through a virtual interface, which the communications component endeavors to make look real. As long as "user" and "system" are concerned, they only need to know in a well defined manner how to invoke functions of the communications component, in order to simulate their mutual direct interaction. In the same sense as we have introduced the relationship "user-system", the communications component is a sub-system to both. It does not have to make any assumption about the relative levels of the two others, as this is only a convention in our mind. If we take the viewpoint of the communications component, the structure rotates 90° to the right, (Fig. 4). "User" and "system" become equal : they are only parties engaged in communications.

Working Size Type Area : 5.5 1/2 x 32 Picas (To be shot at 98%)
Final Size Type Area : 30 x 47 Picas
Final Trim Size : 6 1/2 x 9 7/8 inches (165x251 mm)

Mathematisch Centrum
Tweede Boerhaavestraat 49
Amsterdam, Netherlands

Recap : communications are a lower level transparent to higher levels dependencies. Hence they are symmetrical.

Protocols

Since higher level relative dependencies are immaterial at communications level, we use the term "correspondent" to mean either one of the two levels engaged in communications. They interfere over a virtual interface, but have no direct exchange except through the communications component. The set of rules governing the exchange between the two correspondents has come to be called a "protocol". It is invisible at communications level (2).

Since we are dealing with computers, information exchanged between correspondents are messages. According to the protocol, they may contain ASCII text, control fields, floating point numbers, relocatable binary programs, or whatever. At communications level all messages appear as bit strings. Each message composed by the sender and interpreted by the receiver must conform to the protocol. Consequently there must be some mechanisms in charge of enforcing rules, and rejecting inacceptable messages. These can be viewed as opposite automata, one is the generator of productions of a language, the other is the recognizer. Productions are either accepted or rejected. In the latter, error recovery must be performed. Clearly the sender automaton needs some feedback when it has to stop generating messages and go to error recovery. This means that communications are always bi-directional, and that sender and receiver automata exchange also control information in addition to correspondent messages. This is equivalent to status codes returned to the caller when using a direct interface.

Assuming a reliable communications component, messages are only rejected when they are ill formed, in which case error recovery requires the sender to correct the situation (e.g. when the sender is a person at a console). This should be part of the protocol. But rejection may also be traced to message corruption within the communications component. This is expectable when communications use long distance transmission equipment. Here we have several options :

a) Correcting communications errors may be included as a normal function of the protocol. This means that there is to be some departure from the way correspondents would dialogue, should they interface directly. On the other hand error control encompasses every possible failure end-to-end.

b) A lower level communications protocol may be contrived, to govern exchanges between sender and receiver automata, and correct only communications errors. The corresponding logic

is split into a <u>user protocol</u>, assuming perfect communications, and a <u>communications protocol</u>, <u>nested</u> in the former, (Fig. 5).

c) One may contend that correcting communications errors should logically be put at communications level anyway. Correspondents would only implement a user protocol, assuming perfect communications.

d) But no physical system is perfect. Even though the communications component would correct errors, there would still be unrecoverable or undetected ones. Consequently the user protocol must be designed to make up for rare yet undesirable communications error, lest being occasionally fooled.

e) It may cost the same at user level to detect and correct rare or not quite rare failures. Therefore why put additional mechanisms at communications level ?

f) If communications are extremely reliable, it may be sufficient for users to rely upon deferred control, so reducing the logic embedded in their protocols.

g) The argument can go on and on. Again, there is no possible proof for the best structure, except by experience. Behind all this rationale there is a subjective premise : "I trust my part, not yours". But this is such a common attitude in human life that it should not be discarded as unreasonable in networks. Instead one should identify carefully areas where components are to be legitimately <u>mutually suspicious</u>.

III. DISTRIBUTED MACHINE

A particular network service (e.g. file transfer) is implemented as a set of processors available to users in every host, and defined by a set of rules, (e.g. the file transfer protocol). While a protocol can be viewed as a language, by extrapolation it can also be taken to mean a machine executing programs in that language. Users of that protocol are higher levels in the system hierarchy. Within the network at large they interface the same service with the same language. Conceptually this set of processors make up a single homogeneous machine, whose components happen to be geographically located in various hosts (3). We call such a construction a "distributed machine".

As every abstract view the concept of distributed machine does not change anything in the internal structure, which is a set of autonomous, distant processors. But it helps to present a simpler model at user level, by screening out all considerations not relevant to user processing, such as internal message exchange

between components. The description of a distributed machine can be reduced to an instruction set, and corresponding actions on internal states and input-output (4).

In order to execute an instruction the components of a distributed machine must cooperate. Aside from transferring operands (data), they must keep a consistent set of internal states which constitutes the execution context directing actions to be performed. Thus, part of the communications established between components is devoted to carrying control information for action reports and context updating.

Communications failures are not unlikely, and components may occasionally exhibit erratic behavior without necessarily go dead. In a network environment resorting to manual intervention on several hosts may turn out an impossible task. Consequently, error recovery should be automated as often as possible. This means that execution mechanisms must include back-tracking and retry, somewhat like a few present day computers.

As can be inferred intuitively the critical area in a distributed machine is communications. Not because it is intrisically less reliable or more costly than other functions, but because we are not yet trained to use it as a basic ingredient of computer system architecture.

IV. BASIC COMMUNICATIONS

The one-message case :

A component (S) wants to transmit with certainty a message to another component (R) . No assumptions are made about communications reliability : the message can be delivered correctly within some undefined delay, altered, lost, delivered in several copies, etc... The R-state is unknown to S, since any information received by S about R relates only to some past point in time.

In normal conditions the following sequence of events might take place :

S sends message M1 to R
R receives message M2
R sends M2 back to S
S receives M3
S compares M3 with M1 and finds them equal

At this point can S hold certain that R has received one good copy of M1 ? Not quite, unless we introduce some assumptions :

- M3 really comes from R
- there cannot be an alteration which would turn a bad M2 back into a good M3.

A first conclusion is that messages must contain two addresses. M1 contains R, but also S so that R can send back M2. And M2 contains S and P, so that S can check that M3 comes from R. But this is only valid to the extent that no other receiver can put the R address when sending M2 back to S. Consequently either of the following assumptions must be made :

- the originating address is put in messages off sender's reach
- senders can be trusted for originating addresses.

The previous scheme has the obvious drawback of transmitting the same message twice. We can reduce the loss of bandwidth in having R send back a short checksum of M2, which S can verify for correctness. Then certainty gives way to high probability, actually as high as desirable with a proper checksumming algorithm.

But we save even more if S sends the checksum along with M1, and if R checks M2 correctness. Then a simple empty message (ACKnowledgement) from R to S can be taken to mean correct reception. But more assumptions must be made :

- a receiver can be trusted for checksumming
- incorrect messages are not acknowledged.

We introduce now abnormal conditions. Messages may be lost or delivered after a very long time. At some point S must decide not to wait any longer for ACK. Hence the need for a time-out mechanism. After a number of unsuccessful attempts, S should give up. But what conclusions can be inferred ? Any of the following causes may have been present :

- S to R communications down
- R to S communications down
- R down
- unusual delays

One cannot conclude that R did not receive the message. Actually it may have received all attempts. S ends up with uncertainty.

If R is expecting only one message, all additional copies do no harm. But if it executes a specific action whenever it receives a message, it may not be S intent to have several executions. However there is no way to insure that only one copy will be delivered to R. Even if S never repeats sending, the communications machinery may create duplicates. Consequently R needs some information to be able to inhibit actions for redundant messages. This can be provided in the form of a name attached to the message. All messages bearing the same name will be considered as a

single copy, as far as actions are concerned. But each one is acknowledged, since repetitions may be due to the loss of ACK's.

Once received a message name is kept within the R context. In other words, R creates a name space of received messages in which names must be unique. Before entering a new name it is checked against present ones, and if found the message is a duplicate.

Message set :

When S transmits successive messages to R, and receives ACK's, it has no way to correlate them, if it uses the previous scheme. Counting the number of ACK's and messages is unreliable, due to duplicates and losses. Since R can cope with duplicates, S must insure that every message in the set has been received, therefore acknowledged, at least once. This requires that S keep track of unacknowledged messages and R send back named ACK's.

With this additional feature, message control is performed on each message independently from the others. We only need a simple one-message protocol along with a name space in S and R. A summary of mechanisms introduced so far appears in (Fig. 6).

An obvious weakness of this scheme is the necessity for R to retain every distinct received message name in its own space. Aside from storage and look up overhead, names cannot be indefinitely unique, or they would grow continuously. Duplicates would appear when the name space is exhausted. Consequently a mechanism must be introduced to release dead names and refill the not yet received message name space. Of course both S and R must agree on a common scheme, or else messages would be accepted or discarded untimely.

No matter how S and R draw up their name trade, the communications system might throw in an old message bearing a new name. As this would have an undesirable effect, one relies upon another assumption :

- the communications system can be trusted for delivering messages within well defined limits in time, or not at all.

Let T.min and T.max be the minimum and maximum delay that a message may take to travel from S to R. Then, a message name may be reused after a minimum delay : T.max - T.min, unless repetitions are necessary.

This is a major characteristic of the communications system, since it has a direct impact on the amount of information which makes up the context of S and R. Indeed, knowing the maximum message traffic, a delay corresponds to a certain number of messages.

Let M be the maximum number of messages per unit of time. Assuming S keeps sending messages it must hold for possible repetition copies of unacknowledged ones, viz :

2M T.max

R must hold for possible duplicates the names of last received messages, viz :

M (T.max - T.min)

As can be inferred, the value and the first moment of transit delay in the communications system are to be taken into account in S and R design parameters such as buffer length and time-out. This is an example of cross-level interference within the system structure. In order to meet the time deviation requirement, the communications system may contain a built-in mechanism to exterminate outdated messages. Nevertheless, there is always some marginal probability that a stranded message be delivered out of the regular time span.

At this point it is clear that the reliability of basic communications relies upon statistics. No control mechanism is 100 % safe, it only reduces the undetected error rate to a lower figure. The basic scheme outlined so far makes some assumptions about the good behavior of the correspondents. It also assumes a somewhat predictable transit time in the communications system, but nothing else. Eventually error detection is based on the proper management of a message name space, without eradicating completely failure possibilities. Therefore some room is left for more stringent control depending on the actual environment.

All basic communications protocols stem from the above scheme, which is primarily a simple send-receive-acknowledge sequence for each individual message. Their differences are in the way they manage their message name space. In the following we illustrate a few typical communications protocols.

ISO basic mode :

It is termed : basic mode control procedures for data communications systems (5). As customary, protocols controlling exchanges over a transmission line are called procedures. This protocol is intended for data exchange between a master station (computer) and slave stations (terminals) through multi-drop lines. Using wires allows additional assumptions about the communications system :

- there is no duplicate message
- messages are delivered in the same order as sent.

Every message is acknowledged, and there is only one message in transit at any moment (alternate). But in many cases acknowled-

Working Size Type Area : 35 1/2 x 52 Picas (to be shot at 90%)
Final Size Type Area : 30 x 47 Picas
Final Trim Size : 6 1/2 x 9 7/8 inches (165 x 251 mm)

gements are in the form of special messages, carrying other meanings, (NAK, EOT, ENQ). This is a typical exemple of mixing two levels of protocols, one for message control, the other for syntax correctness in data and device control.

Since there is never more than one outstanding message, they have no name. However acknowledgements may have two different names, alternately, to allow checking for not loosing any. This means that message names (0, 1) are implied by the sender. As long as no two successive ACK's are lost, it allows the sender to differentiate between an ACK or message loss. But not so for the receiver. Therefore, error recovery belongs to the high level of the procedure.

ISO HDLC :

This is again an attempt to mix a message transfer protocol and a component operation protocol (6). Several messages may be sent and acknowledgements deferred. This means that the communications system may have a storage capacity of several messages, as store and forward networks do. But without stating it explicitly, the draft standard assumes sequential communications, i.e. message names are sequence numbers and must be received in order.

The message name space is typically numbers 0 - 15. It is reused in a circular fashion, by releasing names of acknowledged messages. These are only accepted in the correct sequence, therefore any acknowledgment stands for all previously numbered ones. As in our basic scheme, the number they carry relates to the message being acknowledged.

BBN ARPANET :

This one is used for inter-IMP exchange over telephone circuits (7). Thus, only the sender may create duplicates. The message name space is 16. It is managed in 8 pairs. For each pair both names are used alternately. Conceptually this is equivalent to 8 autonomous parallel channels controlled by a 2-name alternate procedure. Since only one message can be in transit on each channel, two names are enough to tell duplicates (8). Acknowledgements are control bits tacked on reverse messages. They carry the acknowledged message name (odd-even bit), and can be sent redundantly, the same way as regular messages.

Message name management :

Like any other space the message name space may be managed using well known techniques depending on environment constraints. E.g. circular space with IN and OUT pointers (HDLC), individual table entries (BBN), dynamic allocation of chunks, degenerate 1-name case (ISO basic mode). It all depends on the size of the space,

i.e. of the storage capacity between sender and receiver, and assumptions made about the communications system (duplicates, losses, sequencing). As a result, designing a protocol is a matter of good balance between reliability and efficiency. There is no ultimate solution. E.g. for packet communications over transmission lines, it seems that a ABN-like procedure is the most efficient (3).

In end-to-end protocols encompassing intricate and obscure inner levels, users may want to take every reasonable precaution to avail themselves with reliable communications, no matter what happens out of their area of control. As we have seen, the basic scheme provides for safe communications, but may require a large name space to cater both for delays and duplicates. A way around this overhead problem consists in "clipping" out the name space to reduce the active area to a narrow <u>window</u>, which can be twitched on at some intervals of time. Since transit delays are not randomly distributed (they are more like Poisson), the window can be set to focus on the main cluster of messages in transit. Thus duplicates are either recognized or rejected as irrelevant. Some proportion of regular messages are also rejected, because they fall off the window, but they are retransmitted. A good balance must be found between space and transmission overhead. One more remark : <u>no</u> message <u>sequencing</u> is required. Sequencing is a particular case when the window of acceptable names is reduced to one.

<u>Error recovery</u> :

A basic communications protocol is nonetheless an instance of a distributed machine, to which general principles apply. Once a correspondent has got out of hand, due to unanticipated failure of critical parts, or a protocol flaw, exchanges are meaningless until a common state of agreement is reinstated. Consequently mechanisms are necessary to direct both automata to a recovery procedure.

One of the correspondents may discover an error condition and decide for itself, but it must insure that the other will follow on. Therefore <u>control messages</u> unrelated to the regular message traffic should be included in the protocol. In case the other component cannot be made to go to recovery state, the healthy correspondent should report an error to its next higher level and reset its state. In any case, recovery may be requested by the higher level if it detects an inconsistency not visible by basic communications (syntax error).

Following an error, correspondents can no longer count on their execution context, which may be inconsistent. But we have seen that this context (name space) is necessary to detect losses and

duplicates in oncoming messages. Consequently all traffic (except error control messages) arriving after an error is detected should be discarded.

Recovery consists in cleaning out the context in " backspacing" up to a point in past traffic which both parties consider safe. It may be a prior checkpoint, or may result from a negotiation according to a recovery protocol. Once consistent contexts have been reinstalled, regular traffic is resumed. Since both correspondents possess the same naming scheme, they can usually restart from the earliest unacknowledged message, unless directed otherwise by the higher level.

Checksumming :

In some cases parity codes are used, but their efficiency is limited in serial transmission. More efficient techniques are based on cyclical redundancy codes (CRC), which can yield an undetected error rate less than 1 bit in 10^{10} with 1000-bit messages and 16-bit CRC.

Due to the substantial time which would be required for software checksumming, this operation is usually done by the I-O hardware. However, in store and forward systems, checksums are stripped off on input, and recomputed on output. In the meantime messagesmay be damaged when they lay unprotected. This is a present weakness, which could be eliminated with an end-to-end checksum.

V. USER-TO-USER PROTOCOLS

Basic level :

When two user programs running in different hosts want to start communicating the least they should be able to do is to exchange simple messages, since any more sophisticated tool requires an initial set up based on messages. This allows bootstrapping whatever protocol they might want to use for private or experimental purpose. Furthermore many simple tasks can be implemented quickly with a few communication primitives, particularly when they are limited to a straightforward dialogue. However, what can they send messages to ?

Subscribers :

It would be unpractical to address messages with names as used locally in each host, because formats would be dependent on particular operating systems. They might also be ambiguous, or unpredictable when a user process is given a name dynamically.

It is much more convenient to create a network-wide name space, and assign subscriber numbers in a structured manner, in the same vein as P.O. Box 62492, Zip 53017.

For convenience in routing messages part of the subscriber name can be a geographical locater, like region, city, host. Without being a compelling feature, it certainly saves time and overhead not to have to look up a general network directory to find out a subscriber location. However "floating" subscribers are perfectly acceptable as long as someone pays for the associated cost.

It is also more convenient to assign permanently subscriber names to correspondents such as human user, terminal, server process, sub-system. As the total number of subscribers is likely to be much larger than the number of simultaneous active correspondents, it may be contended that using too large a name space carries transmission and storage overhead. But if there were a dynamic allocation of subscriber names, some other permanent names would have to be used anyway for correspondents to get in touch initially. On the other hand this problem is already well known in areas such as file systems. Overhead can be reduced by building tables of "active" subscribers, while others reside on secondary storage.

Ports - Links :

Most existing software use the concept of ports to defer till execution time the binding of data streams and I-O devices, or files. Conventional inter-process communications are also modeled after I-O streams. Distributed activities, distant site peripherals or files require cross-network binding. From communications standpoint, it is somewhat similar to connecting two subscribers in the telephone system. Software links should be established between corresponding ports. Again, we have the same problem as in addressing processes. Since port names are local, they are not suited for network naming.

A first solution is to use global network names as a substitute in exactly the same fashion as we have introduced subscriber names. This is the Arpanet way (10). It requires dynamic allocation of global names as anchoring points for port-to-port links. Since these names (sockets) are managed at host level, they require allocation to users either on a permanent or dynamic basis. When it is dynamic, some preliminary protocol must take place to exchange socket names before setting up a link.

Another solution is to use network names having a well defined format, but local to a subscriber. This is the Cyclades way (11). No allocation is necessary, as names are given by users. Furthermore they are only paired at link set up, which does not require prior knowledge of the opposite name. Thus, they are transient

names, because links and ports exist only during execution of a process.

Since a data stream requires some error and flow control, it always has to be matched by a reverse stream. Thus one has the choice to use uni-directional links and set up two for every stream, (Arpanet), or to use bi-directional links (Cyclades).

Sequencing :

This concept applies only to a link. It means that messages are delivered to a port in the same order as they have been sent. This is so obvious in wired communications that not everyone questions the very necessity of that property. In data streams, sequencing is intrisic. But one may think of other types of data transfer for which sequencing is immaterial, as long as no data are altered. E.g. transferring an index-sequential, or direct access file ; or funneling independent transactions toward a common processor ; or collecting statistics ; or transferring a sequential file to a SORT processor, etc...

The alternative is : should all communications be sequential because it is often required, or should that be left up to the user ? Indeed, a multi-node packet switching network is not intrisically sequential, and it takes overhead and restrictions to make it look that way. There is no definite answer yet, as we do not know enough about economics and user experience.

Events :

Like I-O streams, links are subjected to unexpected conditions : accidental closure, message loss ; or they have to relay conditions noticed by a connected process. In other words, in addition to data lines a link should have interrupt lines, so that a signal can jump ahead of the data still in transit. This can be obtained with higher priority messages over a link, or by having a dedicated link for events. In case of erratic behavior of a correspondent, "escape" events should be picked up by the addressee controlling environment. But this can be a dedicated event link.

Fragmentation :

At some point in the communications path there is a maximum message length. The packet switching principle is based on short messages, typically a maximum of 500 to 2000 bits. This is perfectly satisfactory for conversational traffic with short terminals, short transactions, or control messages. This is much too short for data processing, with 500 to 2000-character records, or device handling using blocking factors. Not to mention file transfer. Therefore logical user messages must be broken down into fragments and later on reassembled in the proper order, when they are too long.

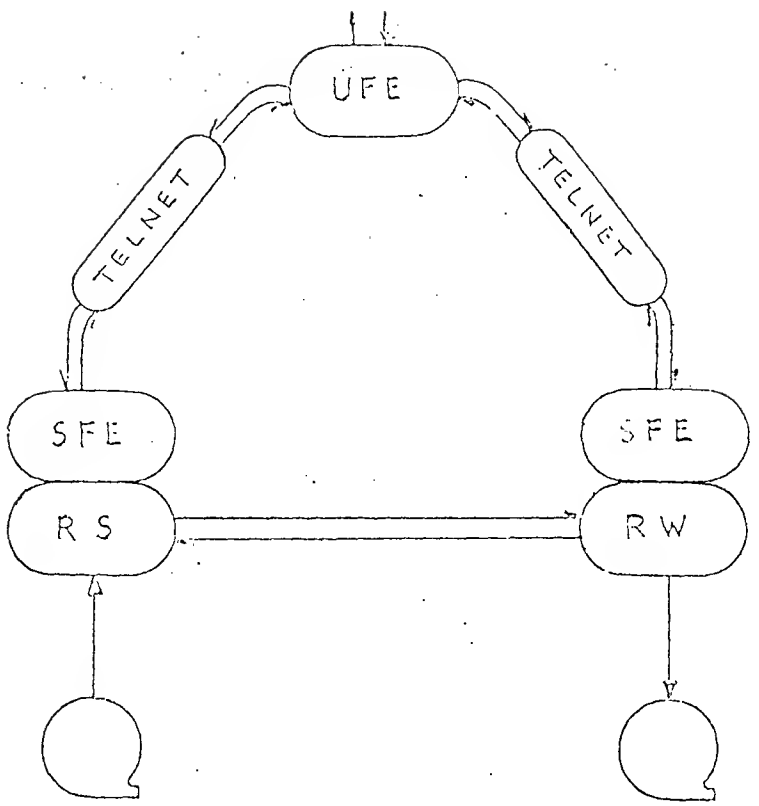


Fig. 7 - File transfer protocol

Whether this operation should be performed only once per message, or at several levels, is purely a matter of economics or technological constraints. A typical packet switching network using mini-computers cannot store very long messages. Furthermore, they would block I-O ports and delay other messages. Therefore only sub-fragmentation can be performed within the communications network. Another level is required anyhow. Then what is the benefit of having fragmentation within packet switching, vs. the resulting constraints ? This is an open question.

A common answer is that it saves on host overhead, notably I-O. But if we look at the way many an operating system handles communications I-O, overhead can be traced back to poor design. Transmission line I-O should be an independent system, possibly wired-in or micro-programmed, instead of a nightmarish kludge of interrupts, clock routines, scheduling, argument checking, register saving, ... except I-O.

If users want to send or receive long messages, they have to provide some corresponding buffers. It may be an appropriate place to fragment and reassemble. Or is it ?

Error control :

If one assumes that lower mechanisms are error free, then error control should be of no concern at user level. But everyone is entitled to his own suspicion. If user protocols do not provide for end-to-end control, at least as an option, likely some users will want to put their own. Bit pattern control may usually be entrusted to lower levels, where it is done quite efficiently. But message naming and acknowledgement may well be organized at user level, because it allows a variety of techniques depending on the type of traffic.

E.g. in a conversational application, control may be based on message syntax and contents : in a D.P. application, it may be done by matching a master file, checksumming a record, adding up fields ; in a program library, there are often specific redundancies which may be taken advantage of.

There may be some inconvenience to handle time-outs as interrupts at user level, because operating systems do not always offer adequate user services in this area. On the other hand system primitives such as SEND, RECEIVE, WAIT, may be equipped with a built-in time-out and an appropriate return status when the user process is awakened.

Likely some error detection and recovery are exercised at lower levels. This might appear redundant, since it usually applies only to some sections of the communications path. But it is not

useless, because it allows quicker detection and reporting of component failures, and recovery at inner levels costs less overhead. Nevertheless, there are always anomalies which cannot be detected or corrected at lower levels, hence an error recovery procedure is always necessary at user level. It may be tailored to the failure rate reported at this level.

However it can be argued whether it is desirable to put error control entirely in user's hands, since he may not be trusted to always do the right thing, and unjustified complaints can deteriorate the network image. Probably a standard error control mechanism should be offered as an alternative, so that casual users have not to make any effort in working one out. This can be a simple acknowledgement of named messages.

Flow control :

For reasons similar to error control, one may contend that users are in the best position to know how much storage they will have available and expected rates of data transfer. Since throughput and overhead are closely tied with flow control, it may be the best (or the worse) choice to leave it up to the user. Flexibility goes with some risks of inappropriate tuning. But it is certainly easier to correct particular user situations than to redesign and phase in a network standard flow control.

It is clear that buffering at lower levels cannot go without its own flow control. The same is true for a communications network. Then one might wonder whether all these layers of control are really useful or whether they result from mutual suspicion and inadequacy. Although the latter may be true, it does not seem to be the fundamental motivation. Flow control is intrisically a resource sharing exercise between different flows and different users. There is nowhere in a system an ideal observation point from which all resources could be apportionned. Depending on level, one sees different flows and resources. E.g. if a user interleaves a diagnostics output stream and a file inquiry onto the same link, he is the only one in a position to control each flow independently and to divide up the link bandwith. This means that flow control, like resource management, is a layered and distributed set of functions. Thus some of it belongs to user level.

VI. SERVER - PROTOCOLS

Server level :

Customarily a server is a non-trivial user system to which commands are adressed directly in some form of high level lingo. It is an abstract machine which converts the traditional user interface into a more sophisticated and spezialized one. Servers are accessed by "users". In this chapter the term user shall mean user of a server.

A file transfer protocol :

We shall take an example with the Arpanet file transfer protocol (12), just to show typical communications aspects of the server level, (Fig. 7). Transferring a file requires the cooperation of two machines, a "reader-sender" (RS) and a "receiver-writer" (RW). But these machines cannot get to work without being fed a program specifying all parameters pertaining to the files being worked on, and the operations to be performed. This program is conveyed into both RS and RW in a conventional command format through TELNET links, a lower level protocol for character string transfer. Since RS and RW do not "understand" character strings, they are assisted with "front-end" (SFE) processors translating commands into an executable form. This is the well known structure of a language interpreter. SFE's are syntax analyzers ; RS and RW are semantic actions.

Commands are sent from a user front-end (UFE) in charge of interfacing with the external user (typically a conversational terminal), and producing commands for SFE's. UFE can be in a third host. After execution commands are acknowledged with status reports flowing back to UFE via the TELNET links.

The file transfer protocol specifies in every detail the relations between UFE and SFE's. It does not specify an external user language, although commands are represented with character strings which could well be a direct console input. Similarly status reports are made up of a (specified) printable code, plus an (unspecified) comment. Here follows a sampling of commands :

class A	class B	class C	
USER	BYTE	RETRIEVE	LIST
PASSWORD	MODE	STORE	ALLOCATE
ACCOUNT	SOCKET	APPEND	RESTART
REINIT	STRUCTURE	DELETE	ABORT
BYE	TYPE	RENAME	

N.B. Spelling is occasionally a short form of the above words.

Class A commands set up a user context. Class B commands set up a file context. Both make up the execution context of the server machine. Class C commands are instructions.

The set of components on (Fig. 7) is nothing but a distributed machine, with its internal communication machinery for synchronizing its processors.

From this observation one might draw some thoughts. Since internal server communications are machine language, is there any subtle advantage to make it look like high level ? Wouldn't it be more efficient to have straight forward communication tools tailored to exchanging contexts, instructions, and operands in a compact form ? On the other hand wouldn't it be rational to have a well defined external user language network-wide ? But that's sociology.

VII. HOST-TO-HOST PROTOCOLS

Host properties :

If one looks for the use made of a host entity within network mechanisms, one finds that it only serves as a geographical pointer. It cannot be allocated, copied, or updated ; it has no attribute ; it is just a name. From communications standpoint, it can be up or down : it is a device. Within hosts there are subscribers, users, servers, files, etc... Hosts are resource containers. Resources are accessible individually from local activities. They can also be accessed remotely from distant activities distributed throughout the network. A communications path is established through a carrier network, and a single device comprising modems, transmission lines, and I-O adapters. For reliability we may want to have two, but that is a separate issue.

Hosts are organized so that parallel activities can execute independently, or in cooperation. Each activity may communicate with other network activities and to that effect calls for communications mechanisms. As seen previously, these are a subscriber name space and a gamut of user protocols. They all must share common physical devices to communicate. Therefore we need a tool to control that multiplexing. Furthermore they do not want to get involved in the details of communications technology. Therefore we need a basic communications facility providing straight forward message transfer.

These are the three basic requirements for a host kit :

- I-O multiplexing
- subscriber name space
- message transfer

Practically, for various extraneous reasons, other functions are also required, such as : - diagnostics, - operator service, - measurement, etc... But these are additional real life aspects not directly implied in network communications.

Network control pool :

As is instantly visible the three functions listed above are quite independent and need not be intertwined. It is even tempting to introduce a tree structure :

a) One I-O multiplexer, with one or several paths to the communications network,

b) An open-ended set of subscriber name spaces,

c) For each subscriber name space, an open-ended set of basic message transfer protocols.

But this may be somewhat arbitrary. A single level for b and c might be more general, since it allows potentially any combination of subscriber and transfer schemes.

Let us call that set of functions a "network control pool" (NCP). Altogether NCP's are a distributed machine establishing a common lower level interface giving access to any host in any other network, provided that matching subscriber and message conventions are implemented.

Design options :

The network control philosophy just brought about is an approach situated at one end of the spectrum of possibilities. Its rationale is to delineate self-contained functions which can work independently assuming only a minimum set of properties about other components, most of which are actually invisible. Sophisticated services are just a juxtaposition of simple functions. Redundancy in implementation can be reduced by a clever design of shared sub-routines.

Another extreme approach is to bury user protocols deep through lower levels, including the communications network. This is typical of one of a kind systems carefully tailored to the requirements of a particular set of applications.

Working Size Type Area : 53 1/2 x 32 1/2 inches (Technical area 90%)
Page Size Type Area : 30 x 17 Picas
Final Trim Size : 6 1/2 x 9 3/4 inches (165x251 mm)

IX. REFERENCES

1 - POUZIN L. - Multi-processor problems and tools for process coordination. NATO Internat. summer school, Copenhagen (Aug. 1970), 30 p.

2 - ELIE M., ZIMMERMANN H. - Vers une approche systématique des protocoles sur un réseau d'ordinateurs, application au Réseau Cyclades. Congrès AFCET (Nov. 1973), 19 p.

3 - THOMAS R.H., HENDERSON D.A. - MC ROSS, a multi-computer programming system. SJCC, (May 1972), 281-293.

4 - POUZIN L. - Network architectures and components. 1st European workshop on computer networks, Arles (Apr. 1973), 227-265, IRIA Ed.

5 - ISO - Basic mode procedures for data communication systems. Doc. 1745.

6 - ISO - High-level data link control procedures. TC 97/SC6, Doc. 794, (Jun. 1973), 33 p.

7 - MC QUILLAN J.M. et al. - Improvements in the design and performance of the Arpa network. FJCC, (Nov. 1972), 741-754.

8 - BARTLETT K., SCANTLEBURY R., WILKINSON P. - A note on reliable full-duplex transmission over half-duplex links. CACM 12, 5, (May 1969), 260-261.

9 - POUZIN L. - Efficiency of full-duplex synchronous data link procedures. Réseau Cyclades, TRA 510, (Jun. 1973), 9 p.

Also available as doc. IFIP/TC6/INWG, Arpa NIC 18255.

10 - CARR C.S., CROCKER S.D., CERF V.G. - Host-host communication protocol in the Arpa network. SJCC (May 1970), 589-597.

11 - POUZIN L. - Presentation and major design aspects of the Cyclades computer network, 3rd data comm. symp., Tampa, (Nov. 1973), 8 p.

12 - NEIGUS N. et al. - File transfer protocol. Arpa network information center, NIC 17759, (Jul. 1973), 50 p.

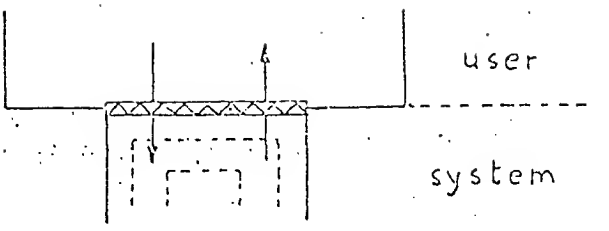


Fig. 1 - Interface

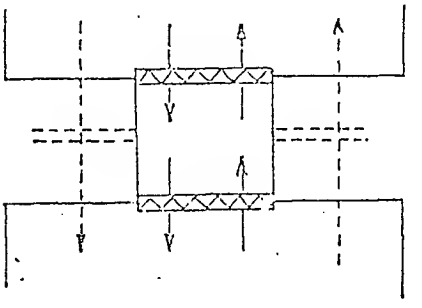


Fig. 2 - Communications interface

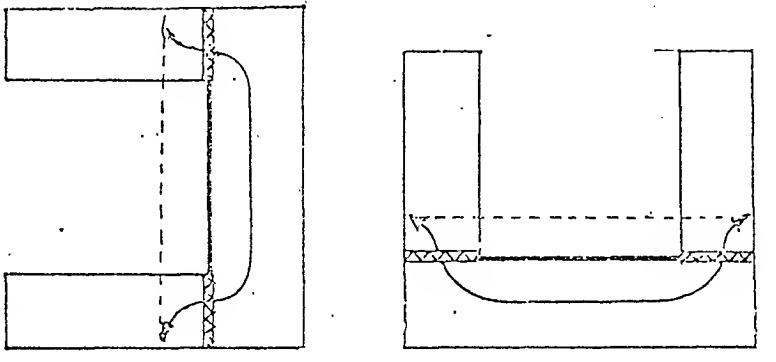


Fig. 3 - Communications sub-system - Fig. 4

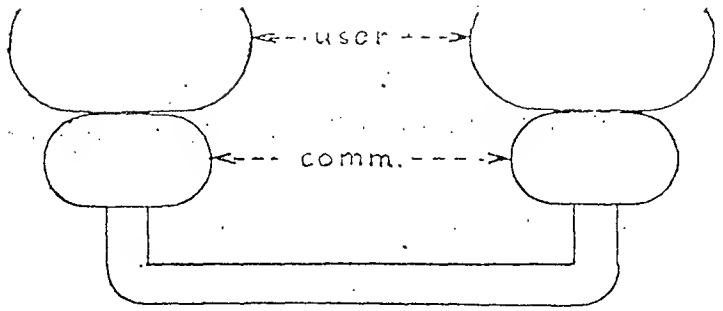
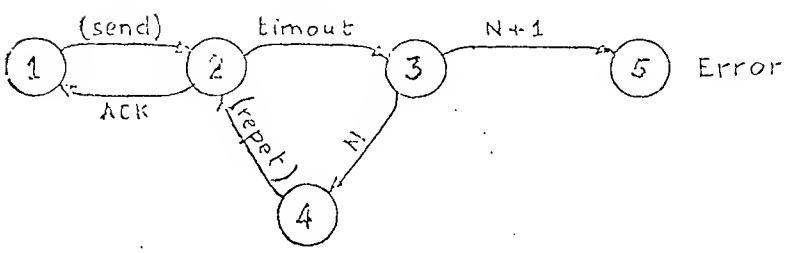
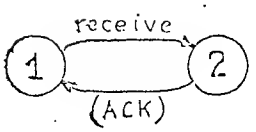


Fig. 5 - Communications



(send) : store message

ACK : cancel message



receive : if name ≠ store
then put in store
take action
else ignore

Fig. 6 - Basic scheme